# CSE 331
# Software Design & Implementation

Hal Perkins

Winter 2023

Debugging

# Administrivia

- HW8 due Thursday – be sure to experiment with all the lecture and section demo programs as well as working on hw8 itself.

- This week:
  - Wednesday: connecting react programs to data
  - Thursday sections: more react/server and hw9

- Reminder: final exam is Tuesday 3/14, 12:30-2:20 for everyone. Same rooms as midterm: GWN 201 (A-I) and GWN 301 (J-Z)
  - Review session: Monday 3/13, 4:30, JHN 102

# A Bug's Life

*defect* – mistake committed by a human

*error* – incorrect computation

*failure* – visible error:  program violates its specification

Debugging starts when a failure is observed

Unit testing

Integration testing

In the field

Goal is to go *from failure back to defect*

# Ways to get your code right

Design & verification

- Prevent defects from appearing in the first place

Defensive programming

- Programming with debugging in mind: fail fast

Testing & validation

- Uncover problems (including in spec.): increase confidence

Debugging

- Find out why a program is not functioning as intended

Testing ≠ debugging

- *test*: reveals existence of problem (failure)
- *debug*: pinpoint location + cause of problem (defect)

# Defense in depth

Levels of defense:

1. Make errors *impossible*
   – Examples: Java prevents type errors, memory corruption
2. Don't introduce defects
   – Correctness: "get things right the first time"
3. Make errors *immediately visible*
   – Examples: assertions, **`checkRep`**
   – Reduce distance from error to failure
4. Debugging is the last level/resort
   – Work from effect (failure) to cause (defect)
   – Use scientific method to gain information
   – Easier to do in modular programs with good specs & test suites

# First defense: Impossible by design

In the language

- Java prevents type mismatches, memory overwrite bugs; guarantees sizes of numeric types, …

In the protocols/libraries/modules

- TCP/IP guarantees data is not reordered
- **`BigInteger`** guarantees no arithmetic overflow

In self-imposed conventions

- Immutable data structure guarantees behavioral equality
- **`finally`** block (or try-with resources) can prevent a resource leak
- Use iteration instead of recursion to avoid stack overflow

Caution: You must maintain the discipline

# Second defense: Correctness

Get things right the first time

- **Think** before you code. Don't code before you think!
- If you're making lots of easy-to-find defects, you're also making hard-to-find defects – don't rush toward "it compiles"

Especially important when debugging is going to be hard

- Concurrency, real-time environment, no access to customer environment, etc.

The key techniques are everything we have been learning:

- Clear and complete specs
- Well-designed modularity with no rep exposure
- Testing early and often with clear goals
- …

These techniques lead to *simpler software*

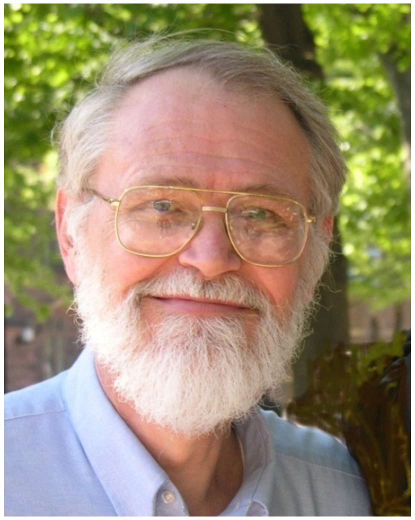
# Strive for simplicity

Sir Anthony Hoare

"There are two ways of constructing a software design:

> One way is to make it so simple that there are obviously no deficiencies, and
>
> the other way is to make it so complicated that there are no obvious deficiencies.

The first method is far more difficult."



Brian Kernighan

"Debugging is twice as hard as writing the code in the first place. Therefore, if you write the code as cleverly as possible, you are, by definition, not smart enough to debug it."

# Third defense: Immediate visibility

If we can't prevent errors, we can try to localize them

Assertions: catch errors early, before they contaminate and are perhaps masked by further computation

Unit testing: when you test a module in isolation, any failure is due to a defect in that unit (or the test driver)

Regression testing: run tests as often as possible when changing code. If there is a failure, chances are there's a mistake in the code you just changed (or the new code is triggering a bug that hadn't been observed before)

If you can localize problems to a single method or small module, you can often find defects simply by studying the program text

# Benefits of immediate visibility

The key difficulty of debugging is to find the defect: the code fragment responsible for an observed problem

– A method may return an erroneous result, even if it is error-free if representation was corrupted earlier

The earlier a problem is observed, the easier it is to fix

– Fail fast: check invariants and assertions frequently
– Don't (usually) try to recover from errors – don't mask them

Don't program in ways that hide errors

– This lengthens distance between defect and failure

# Don't hide errors

```
// pre: x must be present in a
int i = 0;
while (true) {
  if (a[i]==x) break;
  i++;
}
```

This code fragment searches an array **a** for a value **x**

- Value is guaranteed to be in the array
- What if that guarantee is broken (by a defect)?

Temptation: make code more “robust” by hiding errors / not failing

# Don't hide errors

```
// x must be present in a
int i = 0;
while (i < a.length) {
  if (a[i]==x) break;
  i++;
}
```

Now the loop always terminates

- But no longer guaranteed that **a[i]==x**
- If other code relies on this, then problems arise later
- This makes it harder to see the link between the defect and the failure!

# Don't hide errors

```
// x must be present in a
int i = 0;
while (i < a.length) {
  if (a[i]==x) break;
  i++;
}
assert (i!=a.length) : "key not found";
```

- Assertions document and check invariants
  - But are only checked when assertions are enabled
- Abort/debug program as soon as problem is detected
  - Turn an error into a failure
- Unfortunately, we may still be a long distance from the *defect*
  - The defect caused **x** not to be in the array

# Last (inevitable) resort: debugging

Defects happen – people are imperfect

- Industry average (?): 10 defects per 1000 lines of code

Defects happen that are not immediately localizable

- Found during integration testing
- Or reported by user

Cost of an error increases by orders of magnitude during program lifecycle

step 1 – Clarify symptom (simplify input), create “minimal” test
step 2 – Find and understand cause
step 3 – Fix
step 4 – Rerun *all* tests, old and new

# The debugging process

step 1 – *find small, repeatable test case that produces the failure*

- May take effort, but helps identify the defect *and* gives you a regression test
- Do *not* start step 2 until you have a simple repeatable test

step 2 – *narrow down location and proximate cause*

- *Loop*: (a) Study the data (b) hypothesize (c) experiment
- Experiments often involve changing the code
- Do *not* start step 3 until you understand the cause

step 3 – *fix the defect*

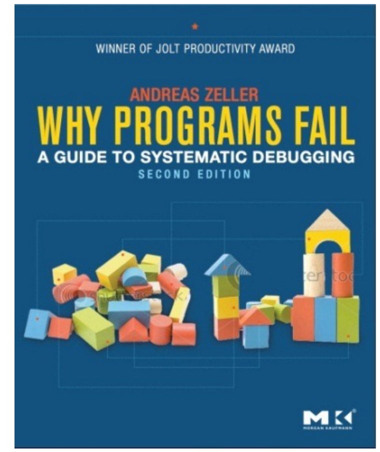
- Is it a simple typo, or a design flaw?
- *Does it occur elsewhere?*

step 4 – *add test case to regression suite*

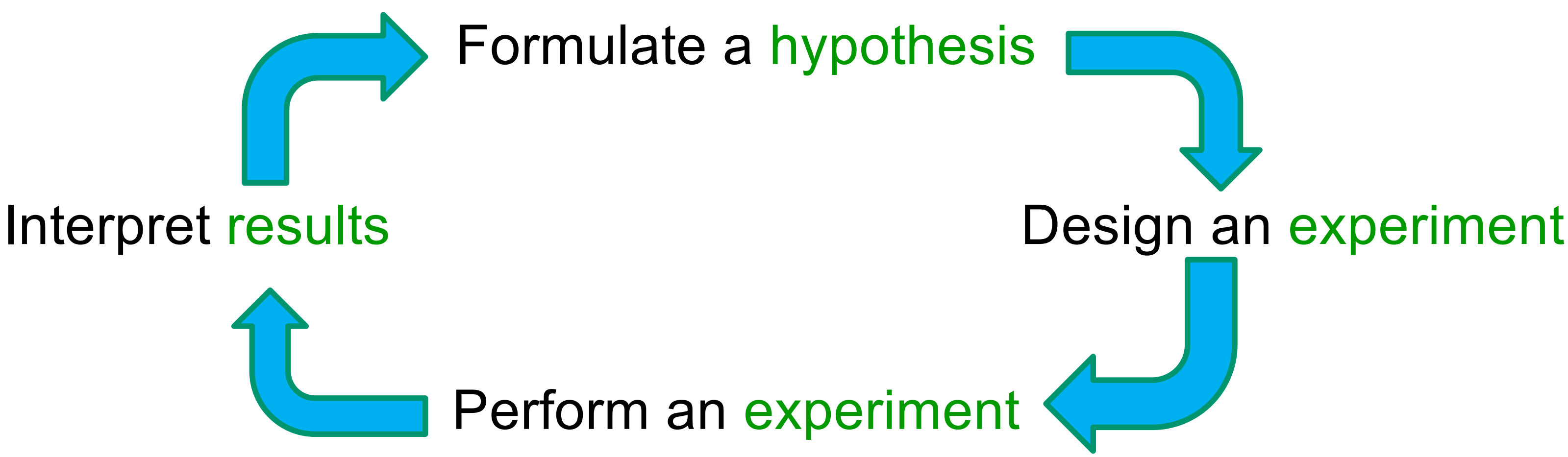
- Is this failure fixed? Are any other new failures introduced?

# Debugging and the scientific method

- Debugging should be *systematic*
  - Carefully *decide* what to do
    - Don't flail!
  - Keep a *record* of everything that you do
  - Don't get sucked into fruitless avenues




- Use an iterative scientific process:



# Example

```
//returns true iff sub is a substring of full
//(i.e. iff there exists A,B such that full=A+sub+B)
boolean contains(String full, String sub);
```

*User bug report:*

It can't find the string **"very happy"** within:

**"Fáilte, you are very welcome! Hi Seán! I am very very happy to see you all."**

Poor responses:

- Notice accented characters, panic about not knowing about Unicode, begin unorganized web searches and inserting poorly understood library calls, …
- Start tracing the execution of this example

Better response: simplify/clarify the symptom…

# Reducing *absolute* input size

Find a simple test case by divide-and-conquer

Pare test down:

*Cannot* find **`"very happy"`** within

**`"Fáilte, you are very welcome! Hi Seán! I am very very happy to see you all."`**

**`"I am very very happy to see you all."`**

**`"very very happy"`**

*Can* find **`"very happy"`** within

**`"very happy"`**

*Cannot* find **`"ab"`** within **`"aab"`**

# Reducing relative input size

Can you find two almost identical test cases where one gives the correct answer and the other does not?

Cannot find **`"very happy"`** within

**`"I am very very happy to see you all."`**

Can find **`"very happy"`** within

**`"I am very happy to see you all.”`**

# General strategy: simplify

In general: find simplest input that will provoke failure

- Usually *not* the input that revealed existence of the defect

Start with data that revealed the defect

- Keep paring it down ("binary search" can help)
- Often leads directly to an understanding of the cause

When not dealing with simple method calls:

- The "test input" is the set of steps that reliably trigger the failure
- Same basic idea

# Localizing a defect

Take advantage of modularity

- Start with everything, take away pieces until failure goes away
- Start with nothing, add pieces back in until failure appears

Take advantage of modular reasoning

- Trace through program, viewing intermediate results

*Binary search* speeds up the process

- Error happens somewhere between first and last statement
- Do binary search on that ordered set of statements

# Binary search on buggy code

```
public class MotionDetector {
    private boolean first = true;
    private Matrix prev = new Matrix();

    public Point apply(Matrix current) {
        if (first) {
            prev = current;
        }
        Matrix motion = new Matrix();
        getDifference(prev,current,motion);
        applyThreshold(motion,motion,10);
        labelImage(motion,motion);
        Hist hist = getHistogram(motion);
        int top = hist.getMostFrequent();
        applyThreshold(motion,motion,top,top);
        Point result = getCentroid(motion);
        prev.copy(current);
        return result;
    }
}
```

no problem yet

*Check intermediate result at half-way point*

problem exists

# Binary search on buggy code

```
public class MotionDetector {
    private boolean first = true;
    private Matrix prev = new Matrix();

    public Point apply(Matrix current) {
        if (first) {
            prev = current;
        }
        Matrix motion = new Matrix();
        getDifference(prev,current,motion);
        applyThreshold(motion,motion,10)
        labelImage(motion,motion);
        Hist hist = getHistogram(motion);
        int top = hist.getMostFrequent();
        applyThreshold(motion,motion,top,top);
        Point result = getCentroid(motion);
        prev.copy(current);
        return result;
    }
}
```

no problem yet

*Check intermediate result at half-way point*

problem exists

# Detecting Bugs in the Real World

Real Systems

- Large and complex (duh ☺)
- Collection of modules, written by multiple people
- Complex input
- Many external interactions
- Non-deterministic

Replication can be an issue

- Infrequent failure
- Instrumentation eliminates the failure
- No printf or debugger

Defects cross abstraction barriers

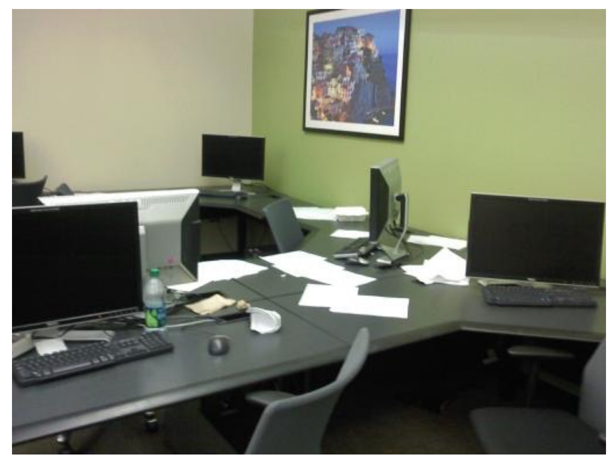
Large time lag from corruption (error) to detection (failure)

# Debugging In Harsh Environments

Failure is non-deterministic, difficult to reproduce

Can't print or use debugger

Can't change timing of program (or defect/failure depends on timing)



Such bugs are more common when users are your testers!

# Heisenbugs

In a sequential, deterministic program, failure is repeatable

But the real world is not that nice…
- Continuous input/environment changes
- Timing dependencies
- Concurrency and parallelism

Failure occurs randomly
- Depends on results of random-number generation
- Hash tables behave differently when program is rerun

Bugs hard to reproduce when:
- Use of debugger or assertions makes failure goes away
    - Due to timing or assertions having side-effects
- Only happens when under heavy load
- Only happens once in a while

# Logging Events

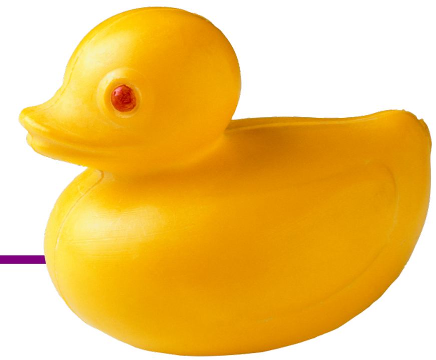
Log (record) events during execution as program runs (at full speed)

Examine logs to help reconstruct the past
- Particularly on failing runs
- And/or compare failing and non-failing runs

The log may be all you know about a customer's environment
- Needs to tell you enough to reproduce the failure

Performance / advanced issues:
- To reduce overhead, store in main memory, not on disk (performance vs stable storage)
- Circular logs avoid resource exhaustion and may be good enough

# More Tricks for Hard Bugs



Rebuild system from scratch, or restart/reboot

- Find the bug in your build system or persistent data structures

Explain the problem to a friend (or to a rubber duck)

Make sure it is a bug

- Program may be working correctly and you don't realize it!

Face reality

- Debug reality (actual evidence), not what you think is true

And things we already know:

- Minimize input required to exercise bug (exhibit failure)
- Add more checks to the program
- Add more logging

# Where is the defect?

The defect is not where you think it is

- Ask yourself where it can not be; explain why
- Self-psychology: look forward to being wrong!

Look for simple easy-to-overlook mistakes first, e.g.,

- Reversed order of arguments: `Collections.copy(src, dest);`
- Spelling of identifiers: **`int hashcode()`**
  **`@Override`** can help catch method name typos
- Same object vs. equal: **`a == b`** versus **`a.equals(b)`**
- Uninitialized data/variables
- Deep vs. shallow copy

Make sure that you have correct source code!

- Check out fresh copy from repository; recompile everything
- Does a syntax error break the build? (it should!)

# When the going gets tough

Reconsider assumptions

- e.g., has the OS changed?  Is there room on the hard drive? Is it a leap year? 2 full moons in the month?
- Debug the code, *not* the comments
  - Ensure that comments and specs describe the code

Start documenting your system

- Gives a fresh angle, and highlights area of confusion

Get help

- We all develop blind spots
- Explaining the problem often helps (even to rubber duck)

Walk away

- Trade latency for efficiency – **sleep!**
- One good reason to start early

# Key Concepts

Testing and debugging are different

- Testing reveals *existence of failures*
- Debugging pinpoints *location of defects*

Use assertions (& checkRep) to turn errors into failures as soon as possible and reduce the distance back to the defect (the actual bug)

Debugging should be a systematic process

- Use the *scientific method*

Understand the source of defects

- To find similar ones and prevent them in the future